Using Genetic Programming for Source-Level Data Assignment to Dual Memory Banks

Using Genetic Programming for Source-Level Data Assignment to Dual Memory Banks

Alastair Murray and Björn Franke

Institute for Computing Systems Architecture
School of Informatics
University of Edinburgh

Abstract. Due to their streaming nature, memory bandwidth is critical for most digital signal processing applications. To accommodate these bandwidth requirements digital signal processors are typically equipped with dual memory banks that enable simultaneous access to two operands if the data is partitioned appropriately. Fully automated and compiler integrated approaches to data partitioning and memory bank assignment have, however, found little acceptance by DSP software developers. This is partly due to the inflexibility of the approach and their inability to cope with certain manual data pre-assignments, e.g. due to I/O constraints. In this paper we build upon a more flexible source-level approach where code generation targets DSP-C [1], using genetic programming to overcome the issues previously experienced with high-level memory bank assignment. We have evaluated our approach on an Analog Devices TigerSHARC DSP and achieved performance gains of up to 1.57 on 13 UTDSP benchmarks.

1 Introduction

Digital signal processors are domain specific microprocessors optimised for embedded digital signal processing applications. The demand for high performance, low power and low cost has led to the development of specialised architectures with many non-standard features exposed to the programmer. With the recent trend towards more complex signal processing algorithms and applications, high-level programming languages (in particular C) have now become a viable alternative to the predominant assembly coding of earlier days. This, however, comes at the price of efficiency when compared to hand-coded approaches [2].

Optimising compiler technology has played a key role in enabling high-level programming for digital signal processors (DSP). Many of the newly developed approaches to code generation for specialised DSP instructions [3], DSP specific code optimisation [4] and instruction scheduling [5] have transitioned out of the research labs and into product development and production.

The situation, however, is different with compiling techniques targeting one of the most distinctive DSP features: dual memory banks. Designed to enable the simultaneous fetch of two operands of, for example, a multiply-accumulate operation, they require careful partitioning and mapping of the data to realise their

full potential. While the dual memory bank concept has found active interest in the academic community this work has not been deployed into production compilers. Instead, DSP specific language extensions of the ISO C language such as DSP-C [6] and Embedded C [7] that shift the responsibility for data partitioning and mapping to the programmer are widely embraced by industry. We believe this is partly due to the fact that fully automated and compiler integrated approaches to memory bank assignment ignore that programmers require control over the mapping of certain variables. E.g., for I/O buffering and tied to a specific bank. Additionally, programmers would frequently like to specify a partial mapping to achieve a certain effect on particular regions of code, and leave the remainder to the compiler. To our knowledge, none of the previously published memory bank assignment schemes allows for this level of interaction.

In this paper we follow a different approach, namely explicit memory bank assignment as a source-level transformation operating on ISO C as input language and generating output in DSP-C. Next to its inherent portability, the advantage of this high-level approach is the ease with which the manual pre-assignment of variables, i.e. coercing them into a specific user-directed bank, can be accomplished. On the other hand, a high-level approach like the one presented in this paper needs to address the difficulty of having to cope with "unpredictable" later code optimisation and generation stages that may interact with the earlier bank assignment.

Where previous approaches, e.g. [8, 9], aim for optimality of the generated partitioning, we have previously shown that an optimal solution according to the standard interference graph model does not necessarily result in the fastest program in practice [1]. Thus a more advanced method of data assignment is required to include the effect of future interactions when choosing an assignment. We therefore hypothesise that a machine learning method will be better able to predict which colourings will be effective, specifically: genetic programming.

Genetic programming (GP) exploits the same evolutionary principles as genetic algorithms. Instead of mutating and breeding strings, however, trees are used to represent functions [10, 11]. Trees are mutated and are allowed to survive by their "fitness", the higher their "fitness" the higher the probability is that they will make it into the next generation. To breed two trees one of each of their sub-trees are swapped and to mutate a tree a sub-tree is replaced with a randomly generated sub-tree. We chose genetic programming due to its high level of flexibility at generating functions for problems which are poorly understood [12].

1.1 Motivation

Efficient assignments of variables to memory banks can have a significant beneficial performance impact, but are difficult to determine. For instance, consider the example shown in figure 1. This shows the `lmsfir` function from the *UTDSP lmsfir_8_1* benchmark. The function has five parameters that can be allocated to two different banks. Local variables are stack allocated and outside the scope of explicit memory bank assignment as the stack sits on a fixed memory bank on our target architecture. On the bottom of figure 1 four of the possible legal

```
void lmsfir(float input[], float output[],
            float expected[], float coefficient[],
            float gain)
{
  /* Variable declarations omitted */

  sum = 0.0;
  for (i = 0; i < NTAPS; ++i) {
    sum += input[i] * coefficient[i];
  }
  output[0] = sum;
  error = (expected[0] - sum) * gain;
  for (i = 0; i < NTAPS-1; ++i) {
    coefficient[i] += input[i] * error;
  }
  coefficient[NTAPS-1] = coefficient[NTAPS-2] +
                         input[NTAPS-1] * error;
}
```

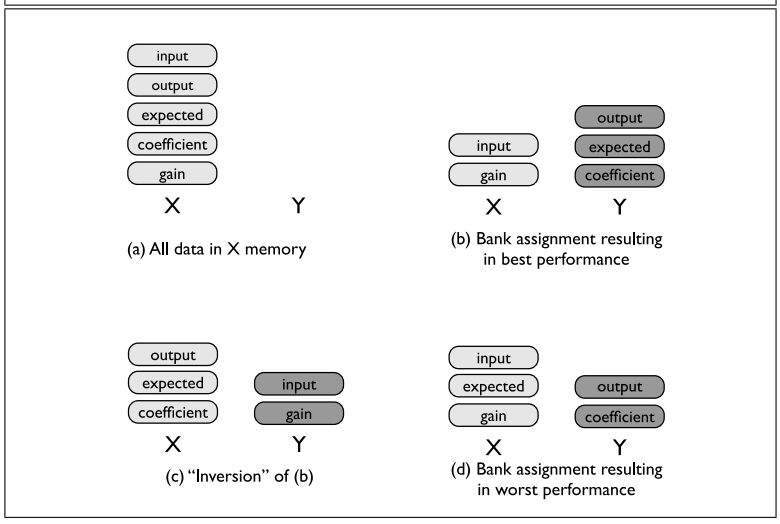


Fig. 1. `lmsfir` function with four memory bank assignments resulting in different execution times.

assignments are shown. In the first case, as illustrated in figure 1(a), all data is placed in the X memory bank. This is the default case for many compilers where no explicit memory bank assignment is specified. Clearly, no advantage of dual memory banks can be realised and this assignment results in an execution time of 100 cycles for our Analog Devices TigerSHARC TS-101 platform. The best possible assignment is shown in figure 1(b), where `input` and `gain` are placed in X memory and `output`, `expected`, and `coefficient` in Y memory. Simultaneous accesses to the `input` and `coefficient` arrays have been enabled and, consequently, this assignment reduces the execution time to 96 cycles. Interestingly, an "equivalent" assignment scheme as shown in figure 1(c) that simply swaps the assignment between the two memory banks does not perform as well. In fact, the "inverted" scheme derived from the best assignment results in an execution time of 104 cycles, a 3.8% slowdown over the baseline. The worst possible assignment scheme is shown in figure 1(d). Still, `input` and `coefficient` are placed in different banks enabling parallel loads, but this scheme takes 110 cycles to execute, a 9.1% slowdown over the baseline.

This example demonstrates how difficult it is to find the best source-level memory bank assignment. Source-level approaches cannot analyse code generation effects that only occur later in the compile chain, but must operate a model generic enough to cover most of these. In this paper we use a refined variable interference graph construction [1] to aid a genetic programming based solution capable of handling complex DSP applications.

The rest of this paper is structured as follows. In section 2 we discuss the large body of related work. Relevant background material is explained in section 3. The source-level memory bank assignment scheme is introduced in section 4, with two different colouring techniques described in sections 5 and 6 before we present our results in section 7. Finally, we summarise and conclude in section 8.

2 Related Work

Gréwal *et al.* used a highly-directed genetic algorithm to provide a solution to dual memory bank assignment [13]. They used a constraint satisfaction problem as a model, with hard constraints such as not being able to exceed memory capacity, and soft constraints such as not wanting interfering variables in the same memory. The genetic algorithm is then used to find the optimal result in terms of this model. This use of machine learning does not actually learn trends regarding the problem, but is more akin to solving the constraint satisfaction problem by brute force as it is rerun for every instance of the problem. Given the high computational cost of running a genetic algorithm it seems undesirable to include one into the run-time of the compiler. Additionally, due to technical limitations this method was only evaluated on randomly generated synthetic benchmarks.

Several authors have proposed integer linear programming solutions. Initially Leupers and Kotte described a method [8] that modelled the interference graph between variables as an integer linear program and tries to minimise total in-

terferences. This approach worked on the compiler IR after the back-end has been run once, allowing it access to very low-level scheduling and memory access information. Another approach by Ko and Bhattacharyya uses synchronous data flow specifications and the simple conflict graphs that accompany such programs [14]. They used an integer linear program to find an assignment to memories, but for all benchmarks that the techniques were evaluated against there exists a two-colouring, so the technique is not demonstrated to work on hard problems. More recently Gréwal *et al.* described a more accurate integer linear programming model for DSP memory assignment [9]. The model described here is considerably more complex than the one previously presented by Leupers and Kotte [8] but provides greater improvements.

Sipkovà describes a technique [15] that operates at a higher-level than the previously described methods. It performs memory assignment on the high-level intermediate representation, thus allowing the assignment method to be used with each of the back-ends within the compiler. The problem is modelled as an independence graph and the weights between variables take account of both execution frequency and how close the two accesses are in the code. Several different solutions, based on a max-cut formulation, were proposed. Unfortunately, this paper does not address any of the issues created by assigning data to memories at a high-level. So it is not clear the technique is as portable as is claimed, nor that it is taking full advantage of the dual-memory capability.

We, Murray and Franke, have previously described an alternative high-level technique [1] that works at the source-level. Specifically, ISO-C is taken as input and DSP-C (see section 3.2) is produced as an output. This paper evaluates the effects of high-level assignment on the effectiveness of integer linear programming (ILP) based data assignment and finds that the ILP solution suffers from inexact interference data. This has the effect of causing the ILP "optimal" solutions to represent a range of results (rediscussed in section 5). An alternative probabilistic data assignment approach called soft colouring was proposed, it achieved a similar speed-ups to the ILP approach but with a much lower run-time.

Finally, Stephenson *et al.* use genetic programming in compilers [16, 12], though for completely different purposes than dual memory bank assignment. They use genetic programming to generate heuristics for priority functions related to hyperblock formation, register allocation and data prefetching. Although many of the results they report are due to evaluating the heuristics on their own training data they also present results for separate test benchmarks. They were able to improve on the existing heuristic in a mature compiler by 9% on average for hyperblock formation across many SPEC benchmarks, demonstrating the potential of genetic programming.

3 Background

3.1 Dual Memory Banks

Typical digital signal processing operations such as convolution filtering, dot product computations and various matrix transformations make intensive use

of *multiply-accumulate (MAC)* operations, i.e. computing the product of two numbers and adding the product to an accumulator.

Digital signal processors are application-specialised microprocessors designed to most efficiently support digital signal processing operations. Among the most prominent architectural features of DSPs are support for MAC operations in the instruction set and dual memory banks that enable simultaneous fetching of two operands. Provided the data is appropriately partitioned across the two memory banks this effectively doubles the memory bandwidth and ensures efficient utilisation of the DSP datapath.

3.2 DSP-C and Embedded C

DSP-C [6] and its later extension *Embedded C* [7, 17] are sets of language extensions to the ISO C programming language that allow application programmers to describe the key features of DSPs that enable efficient source code compilation. As such, DSP-C includes C-level support for fixed point data types, circular arrays and pointers, and, in particular, divided or multiple memory spaces.

DSP-C uses address qualifiers to identify specific memory spaces in variable declarations. For example, a variable declaration like `int X a[32];` defines an integer array of size 32, which is located in the X memory. In a similar way, the address qualifier concept applies to pointers, but now up to two address qualifiers can be provided to specify where the pointer and the data it points to is stored. For example, the following pointer declaration `int X * Y p;` describes a pointer `p` that is stored in Y memory and points to integer data that is located in X memory. For unqualified variables a default rule will be applied (e.g. to place this data in X memory).

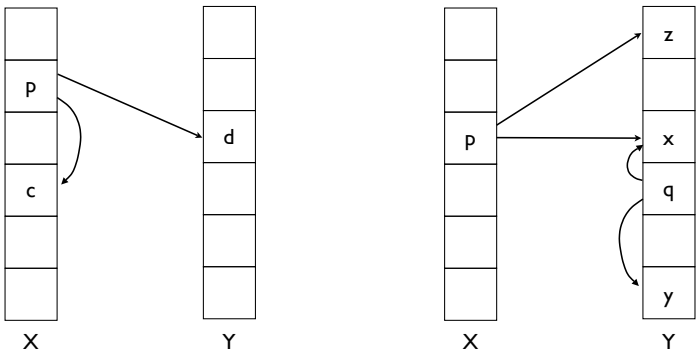
4 Methodology

Our memory bank assignment schemes comprises the following the stages:

1. **Group Forming.** During this stage groups of variables that must be allocated to the same memory bank, due to pointer aliasing, are formed.
2. **Interference Graph Construction.** An edge-labelled graph representing potential simultaneous accesses between variables is constructed.
3. **Colouring of the Interference Graph.** Finally, the nodes of the interference graph are coloured with two colours (representing the two memory banks) such as to maximise the benefit from simultaneous memory accesses.

Of these three stages only stage one is critical for correctness, whereas approximations are acceptable for stages two and three. I.e. an inaccurate interference graph or a non-optimal colouring still results in correct code that, however, may or may not perform optimally.

For both the colouring methods used here the first two stages are the same, for the third stage either the integer linear programming colourer gets dropped in, or the colourer produced by genetic programming. Note that unlike in other



(a) Incompatible pointer assignments. (b) Pointer induced variable grouping.

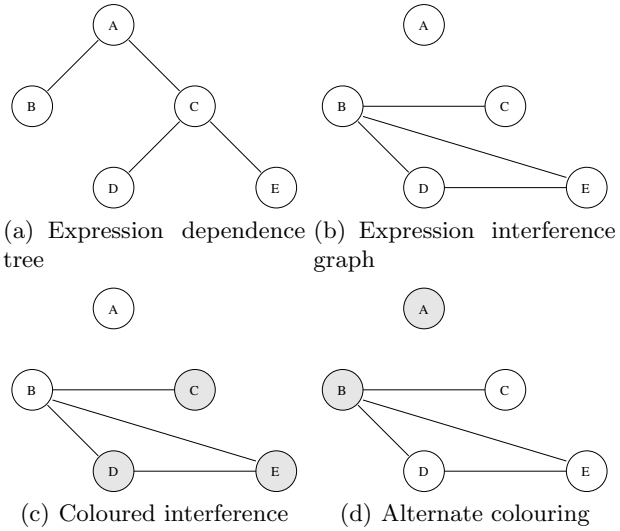
Fig. 2. Incompatible pointer assignments and pointer induced grouping.

genetic approaches to the memory assignment problem [13] the genetic algorithm is trained off-line, so only the function produced the genetic programming will be run during colouring.

4.1 Group Forming

Group forming is the first stage in our memory bank assignment scheme. It is based on pointer analysis and summarises those variables in a single group that arise through the *points-to* sets of one or more pointers. All variables in a group must be allocated to the same bank to ensure type correctness of the memory qualifiers resulting from our memory bank assignment.

Figure 2 illustrates this concept. In figure 2(a) the pointer p may point to c or d. However, c and d are stored in memory banks X and Y, respectively. This eventually causes a conflict for p because both the memory bank where p is stored and the bank where p points to must be statically specified. Thus, p must only point to variables located in a single bank. A legal assignment would place c and d in the same bank as a result of previous grouping. This grouping is shown in figure 2(b) for two pointers p and q. In this example p may point at variables x and z at various points in the execution of a program and, similarly, q is assumed to point at x and y. Grouping now ensures that x and z are always stored in the same bank (due to p), and also x and y (due to q). By transitivity, x, y and z have to be placed in the same memory bank. Note that p and q themselves can be stored in different memory banks, only their targets must be grouped and located in a single memory bank. The algorithm to calculate these groups is described in our previous paper [1].



(a) Expression dependence tree (b) Expression interference graph

(c) Coloured interference (d) Alternate colouring

Fig. 3. Mapping dependances to potential interferences.

4.2 Interference Model

To be able to effectively assign groups of variables to memory banks it is necessary to build an interference graph that represents the memory accesses in the program. This is done statically by taking the dataflow dependence graph for each expression and marking each pair of memory or variable accesses with no dependence between them as potentially interfering (see figure 3). This represents cases where loads or stores could be scheduled in parallel. Each of these potential interferences is given a weight that is equal to the estimated number of times that the expression will be executed. This estimate is determined by calculating each loop's iteration count (or using a constant value if the exact count can not be statically determined), and assuming all non-loop branches are taken with 50% probability. The estimated call count for each function is also calculated this way by estimating how many times each call site is executed. This variable interference graph is then reduced to a group interference graph using the previous assignments. This approximate information is sufficient for determining which groups of variables are the most important.

For this interference graph to be usable by genetic programs it must be reduced further. This is because a genetic program will be more effective if it has a fixed number of features to operate on, rather than an arbitrary number of neighbours. As a genetic program will only be colouring one node of the graph at a time, we can exploit this and provide a "view" of the graph for a specific node and as this "view" will be generated on demand it works well with partially coloured graphs. This is essential as the genetic program colours the nodes in some sequential order, not simultaneously, and will need to consider both already-coloured and not-yet-coloured neighbours.

To reduce the number of nodes to a fixed number, we can first discard all nodes that do not interfere with the current node. Then we can recognise that each of these nodes must belong to one of three classes: not yet assigned to a memory bank, assigned to memory bank X or assigned to memory bank Y. So the remaining nodes can be collapsed down to three nodes, with all interference information being aggregated so as each collapsed node accurately represents the sum of its constituents.

5 Integer Linear Program Colouring

A reference Integer Linear Programming (ILP) colouring approach that is approximately equivalent to the model by Leupers and Kotte [8] is implemented. This model is fully described in our previous paper [1].

Although the ILP method finds the "optimal" solution, the colouring found is not necessarily truly optimal though, it is only an optimal solution in terms of the interference graph. This ILP model is based on the model described in the Leupers and Kotte paper [8], their model used an interference graph built after the back-end of the compiler had run so it is a reasonably accurate model of the potential parallelism in the program. However, in our technique we build the interference model based on the program's source-code, the entire target compiler still has to be run on the program after variables have been assigned to memory banks.

Building the interference graph at a high-level also means that the problem is less constrained, this means that there may be many optimal solutions to the ILP model. For example, if a node is completely disconnected in the interference graph then the score to be maximised by the linear solver will be the same whichever memory bank that group of variables is assigned to.

ILP solvers generally work by first reducing as much of the program to a non-integer linear problem that can be solved quickly and then using a branch-and-bound technique to solve what remains. If there are multiple optimal solutions then they may only be found during the branch-and-bound stage, where it is possible to keep on searching even after an optimal solution has been found. However, in the process of reducing the integer problem to a non-integer one, many of the alternate optimal solutions may be lost and there will be fewer solutions for the branch-and-bound technique to find. An example of this can be seen in figure's 1(b) and 1(c). These two assignments are equivalent in the ILP model, switching between them just results in a complete inversion of the interference graph, so if 1(b) is optimal in the ILP model then so is 1(c). When running on the hardware, however, we find that 1(c) does not perform as well.

6 Genetic Program Colouring

We want use the genetic programming library to produce a function that will colour every node in an interference graph. It is unrealistic to expect genetic programming to produce a function that will return a complete graph colouring,

Feature	Description
Parallel Interference	The no. of interferences at an immediate parallel level.
Para. Interfere. Accuracy	The estimated accuracy of the parallel interferences.
Expression Interference	The no. of interferences at an expression level.
Expr. Interfere. Accuracy	The estimated accuracy of the expression level.
Symbols: Aggregate	No. of aggregate symbols (e.g. structs) in group.
Symbols: Arrays	No. of array symbols in group.
Symbols: Pointers	No. of pointer symbols in group.
Symbols: Scalar	No. of scalar symbols in group.
Type: Integer	No. of integer symbols in group (e.g. an array of ints).
Type: Float	No. of floating point symbols in group.
Type: Complex	No. of non-numerical symbols in group (e.g. a void pointer).
Size	Total no. of bytes occupied by all variables in this group.
Size Accuracy	The estimated accuracy of the size of this group.

Table 1. Program features available to a genetic program.

so we instead colour the graph one node at a time. A view of the graph is given from the perspective of the current node from the interference graph, as described in section 4.2. The three neighbour nodes are constructed (assigned to X, assigned to Y, not yet assigned). Now we run the the function produced by the genetic programming library (which will be an entirely random tree initially) twice for each node, once saying if this node is assigned to X then what score would you give it, and the same for Y. This is done by mapping the X and Y nodes to *Same Colour* and *Different Colour* node – and vice versa for testing a Y assignment. We assign the node to the colour with the higher score. For each function we repeat this process on every benchmark in our training data.

Once all the nodes on all the benchmarks have been assigned a colour we can assign a fitness to the function. Using an table of previously generated exhaustive results we look up the performance of this colouring. If it is equivalent to the best possible colouring the function is given a fitness of 0.0 (best possible). If it is equivalent to the worst colouring it is given a fitness of 1.0 and results in-between are assigned a fitness proportionally. The overall fitness of a function is its average fitness across all benchmarks in the training data. Functions with a better fitness have a higher chance of being used in breeding and of surviving into the next generation.

The benchmarks were evaluated using leave-one-out cross-validation, so for each of the 13 benchmarks used for evaluation the GP colourer was trained on the other 12 benchmarks and then tested against the 13th. This ensures the results are representative of how the colourer will perform on an unseen program.

The full-set of features used are described in table 1, this set of features is replicated for each of the X, Y and *unassigned* nodes. The full set of mathematical and logical operators available are described in table 2. The genetic programming library produces trees consisting of these nodes, it ensures that all nodes have the correct number of children. The only data-type is a floating point number. Because every function produced by the genetic programming library

Function	No. Inputs	Description
Add	2	$A + B$
Sub	2	$A - B$
Mul	2	$A * B$
Div	2	$A \div B$
Sqrt	1	$\sqrt{A}$ (returns 0.0 for negative inputs)
Abs	1	$\|A\|$
Max	2	$\max A, B$
EQ	4	`if (A == B) { C } else { D }`
GE	4	`if (A >= B) { C } else { D }`
GT	4	`if (A > B) { C } else { D }`
LE	4	`if (A <= B) { C } else { D }`
LT	4	`if (A < B) { C } else { D }`
Const	0	`random` $(0 \leq X < 1000)$

Table 2. Functions available to a genetic program.

is guaranteed to be valid, and every colour assignment possible is valid we are assured that every function produced will be accurately evaluated.

To try and improve the performance of the functions produced by genetic programming, three modifications were attempted. The modifications are described here, their effects are described in section 7.

Firstly changed the way the evolved program was used by considering the order in which the nodes are coloured, two variations were attempted. In the first variation we sorted the nodes according to the sum of their interferences, so the most critical nodes are coloured first. In the second variation, instead of just running the generated function on one node we ran it on every uncoloured node in the graph. We then coloured the node that was assigned the highest score overall, this is effectively letting the generated function pick the order in which to colour the nodes. This affects time it takes to perform the colouring, for a graph with n nodes it takes $O(n)$ time to produce a graph view for a given node, with pre-ordered nodes $O(n)$ nodes are evaluated resulting in a colouring time of $O(n^2)$. For a self-ordering method $O(n^2)$ nodes are evaluated resulting in a colouring time of $O(n^3)$.

Secondly we changed how the evolved program was evaluated by adding a size penalty heuristic to the fitness function. Functions with fewer than 10 nodes got no penalty, functions with more than 50 got a penalty of 1.0 (a very large penalty), sizes between these are penalised proportionally. The aim of doing this was to stop huge functions highly specialised to the training data from being generated.

Thirdly we reduced the amount of potentially extraneous information available to the evolved program, by attempting to perform the assignment without using the *unassigned* node. The idea behind this was that nodes that are not yet assigned to a memory bank do not effect immediate assignment decisions – it wasn't clear if the functions would be able to use the information to "plan ahead" or if it was just noise.

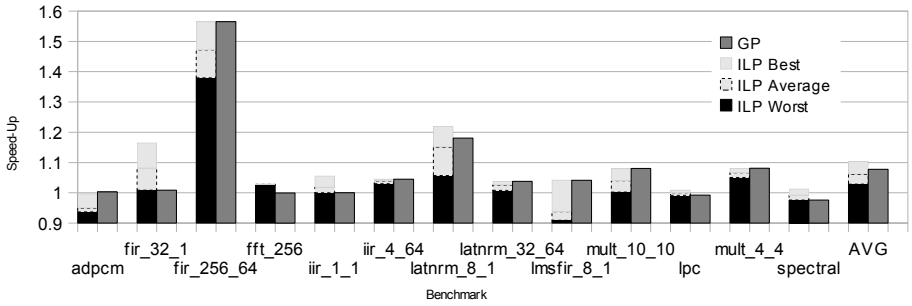


Fig. 4. A comparison of the range of ILP solutions against the GP solutions.

7 Experimental Evaluation

7.1 Platform and Benchmarks

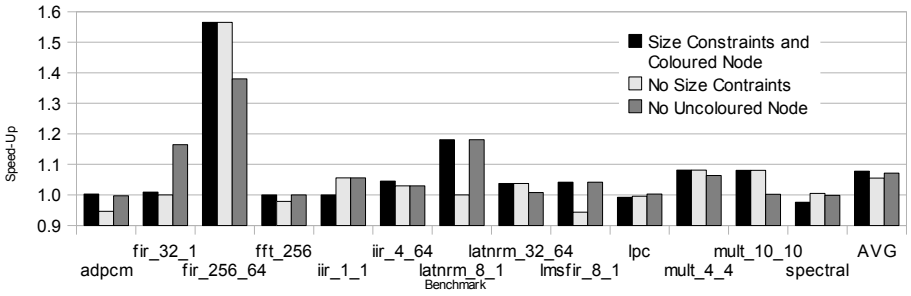
We implemented our source-level C to DSP-C compiler using the SUIF compiler framework [18]. The C program is converted into the SUIF intermediate format which is then annotated with aliasing information using the SPAN tool [19]. We use this information to form groups of variables as described in section 4.1 and output DSP-C with group identifiers in place of memory qualifiers. The C preprocessor may be used to assign a group of variables to a specific memory bank according to the generated group to memory bank mapping.

Both the ILP colourer and the GP colourer are implemented in Java. The ILP colourer makes use of the *lp_solve* [20] library, which is implemented as a native binary, with the default pre-solve and optimisation settings. The GP colourer uses the *ECJ* [21] package to evolve and execute genetic programs. The evolutionary settings used were ECJ's default Koza [11] parameters, with a population of 1024 and 50 generations. Additionally a small amount of elitism is used, the best 2 functions from each generation always survive into the next.

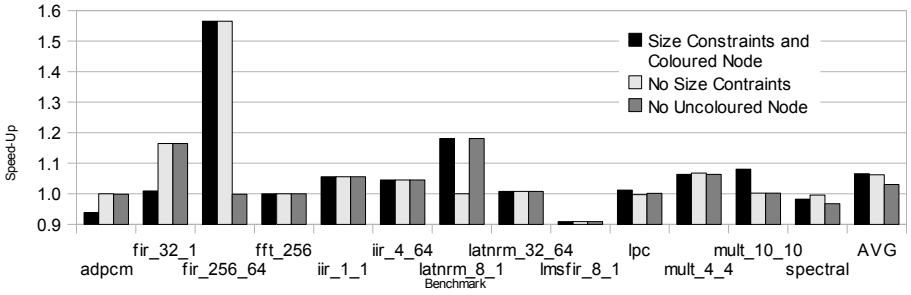
The colourings were done on a Linux system with two dual-core 3.0GHz Intel Xeon processors and 4GB of memory. The experiments where run on an Analog Devices TigerSHARC TS-101 DSP operating with a clock of 300MHz, the DSP-C programs were compiled using the Analog Devices VisualDSP++ compiler. We evaluated our technique using the UTDSP benchmark suite [22]. Each colouring was only run once as the TigerSHARC's static pipeline and lack of cache results in deterministic hardware.

7.2 Results

We compared the GP colourer against the ILP colourer. Figure 4 shows the speed-up achieved by the best GP colourer (nodes may be coloured in any order, constrained size of genetic function and has access to information on uncoloured nodes) against the range of ILP results. The 'ILP Best' and 'ILP Worst' bars in the figure correspond to the highest and lowest ILP speed-ups, relative to



(a) Self-ordering GP.



(b) Pre-ordered GP.

Fig. 5. A comparison of the different modes of operation for the GP.

the performance of ISO C, in the set of equivalent ILP solutions found per benchmark. The 'ILP Average' bars represent the average speedup of these sets. There is not a range of GP results for each benchmark as each genetic program only ever returns one colouring for a given input program. The average speed-up achieved by the GP colourer is 1.078, which although not as good as the upper range of the ILP colourer's potential it is much higher than the lower end. The GP colourer achieves 65.7% of the performance available in ILP's range of potential performance (speedups of 1.030 to 1.103). This is significant because it means that given the generally uniform distribution of ILP performance across it's range of potential results (demonstrated by the average speedup across a set of ILP solutions generally being equidistant between the best and worst solutions), the GP colourer will out-perform ILP 65.7% of the time. Other points of note are that the GP colourer only results in a slow-down for two benchmarks (*lpc* and *spectral*), whereas the ILP colourer may result in slow-downs for four benchmarks. Additionally, the ILP colourer always results in a slowdown for *adpcm* but the GP colourer manages to obtain a small speed-up.

In the process of developing the GP colourer different methods were experimented with. Here we compare three variations described in section 6 to the method that was found to be best. First pre-ordered vs self-ordered nodes, the general trend is that letting the GP colour the nodes in any order is almost always better than arranging the order beforehand. If keeping the other variants

fixed then average speed-ups of 1.078 and 1.065 are achieved respectively, this trend holds for the other combinations of the variants.

Secondly, we penalised the fitness of larger functions. It was found that this penalty improved performance, the effects of eliminating it may be seen in the first and second columns of each benchmark in figures 5(a) and 5(b).

Thirdly, we eliminated the uncoloured node from the reduced interference graph (see section 4.2). The effects of this may be seen by comparing the first and third columns of each benchmark in figures 5(a) and 5(b). In most cases this made little difference. A few benchmarks, however, suffered without this node so eliminating it slightly reduces the performance of the colourer on average.

The time taken to do the colouring for these benchmarks is trivial. Even if including the time to perform alias analysis, the only benchmarks to take longer than one second to colour are *adpcm* and *spectral* (for both the ILP and GP colourers, as alias analysis dominates the run-time for these benchmarks). If, however, we take the *adpcm* program and modify the code so as all *automatic* variables are made into global variables, we can obtain a semi-synthetic program that has a much larger number of groups to colour. Once alias analysis has been performed, which takes 7 seconds, the ILP colourer takes a further 1.5 hours to find a solution. The GP colourer takes under a second, demonstrating the benefit of its polynomial run-time over the ILP colourer's exponential run-time.

8 Summary and Conclusions

We have presented a method for performing dual memory bank assignment at the source-level, using a C to DSP-C compiler. We have demonstrated an assignment technique that performs more predictably than ILP colouring, has a lower execution time and is able to find a better solution than the "optimal" ILP colourer 65.7% of the time. We evaluated our technique on the *UTDSP* benchmark suite where we achieved a 7.8% speed-up on average, out of an absolute maximum of 10.4%.

Our technique may be easily introduced to an existing DSP tool-chain due to operating at the source-level. The ability to train the tool offline means that a large suite of benchmarks could be used to train the tool, resulting in good performance with an extremely low execution cost.

References

1. Murray, A., Franke, B.: Fast source-level data assignment to dual memory banks. In: Proceedings of the 11th International Workshop on Software and Compilers for Embedded Systems (SCOPES '08). (March 2008) 43–52
2. Frederiksen, A., Christiansen, R., Bier, J., Koch, P.: An evaluation of compiler-processor interaction for DSP applications. In: Proceedings of the 34th IEEE Asilomar Conference on Signals, Systems, and Computers. (2000)
3. Bhattacharyya, S., Leupers, R., Marwedel, P.: Software synthesis and code generation for signal processing systems. IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing **47**(9) (2000)

4. Leupers, R.: Novel code optimization techniques for DSPs. In: Proceedings of the 2nd European DSP Education and Research Conference. (1998)
5. Timmer, A., Strik, M., van Meerberger, J., Jess, J.: Conflict modelling and instruction scheduling in code generation for in-house DSP cores. In: Proceedings of the Design Automation Conference (DAC). (1995)
6. ACE: DSP-C, an extension to ISO/IEC IS 9899:1990. Technical report, ACE Associated Compiler Experts bv (1998)
7. JTC1/SC22/WG14: Programming languages - C - extensions to support embedded processors. Technical report, ISO/IEC (2004)
8. Leupers, R., Kotte, D.: Variable partioning for dual memory bank DSPs. In: Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP '01). Volume 2. (May 2001) 1121–1124
9. Gréwal, G., Coros, S., Morton, A., Banerji, D.: A multi-objective integer linear program for memory assignment in the DSP domain. In: Proceedings of the IEEE Workshop on Memory Performance Issues (WMPI '06). (February 2006) 21–28
10. Cramer, N.L.: A representation for the adaptive generation of simple sequential programs. In: Proceedings of the International Conference on Genetic Algorithms and their Applications (ICGA85). (1985) 183–187
11. Koza, J.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. The MIT Press (1992)
12. Stephenson, M., Martin, M., O'Reilly, U.M., Amarasinghe, S.: Meta optimization: Improving compiler heuristics with machine learning. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implemenation (PLDI '03). (June 2003) 77–90
13. Gréwal, G., Wilson, T., Morton, A.: An EGA approach to the compile-time assignment of data to multiple memories in digital-signal processors. SIGARCH Computer Architecture News **31**(1) (March 2003) 49–59
14. Ko, M.Y., Bhattacharyya, S.S.: Data partioning for DSP software synthesis. In: Proceedings of the International Workshop on Software and Compilers for Embedded Systems (SCOPES '03). (September 2003) 344–358
15. Sipkovà, V.: Efficient variable allocation to dual memory banks of DSPs. In: Proceedings of the 7th International Workshop on Software and Compilers for Embedded Systems (SCOPES '03). (September 2003) 359–372
16. Stephenson, M., O'Reilly, U.M., Martin, M.C., Amarasinghe, S.: Genetic programming applied to compiler heuristic optimization. In: Proceedings of the 6th European Conference on Genetic Programming. (April 2003)
17. Beemster, M., van Someren, H., Wakker, W., Banks, W.: The Embedded C extension to C. http://www.ddj.com/cpp/184401988 (2005)
18. Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.M., Tjiang, S.W.K., Liao, S.W., Tseng, C.W., Hall, M.W., Lam, M.S., Hennessy, J.L.: SUIF: An infrastructure for research on parallelizing and optimizing compilers. SIGPLAN Notices **29**(12) (December 1994) 31–37
19. Rugina, R., Rinard, M.: Pointer analysis for multithreaded programs. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implemenation. (May 1999) 77–90
20. : lp_solve package. http://lpsolve.sourceforge.net/5.5/ (2008)
21. : ECJ package. http://cs.gmu.edu/ eclab/projects/ecj/ (2008)
22. Lee, C.G.: UTDSP benchmark suite. http://www.eecg.toronto.edu/ corinna/DSP/infrastructure/UTDSP.html (1998)